Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation

Hartmut Ehrig [1] Karsten Ehrig [2]

Technical University of Berlin, Germany

Abstract

In this paper we give an overview of formal concepts for model transformations between visual languages based on typed attributed graph transformation. We start with a basic concept where visual languages are defined by attributed type graphs only and model transformations by basic typed attributed graph transformation systems. We continue with different kinds of extensions of the basic concepts taking into account application conditions, constraints, generating graph grammars and operational semantics. The main aim is to discuss formal correctness criteria for model transformations including syntactical correctness, functional behavior and semantical correctness.

Key words: model transformation, overview, formal concepts, typed attributed graph transformation, graph transformation

1 Introduction

With the *Model-Driven Architecture (MDA)* [OMGb] the *Object Management Group (OMG)* has defined a standard for software development based on modeling and automated mapping of models to implementations. For defining models and metamodels the OMG has established the well known standards *Meta-Object Facility (MOF)* [OMGa] and *Unified Modeling Language* [UML] which provide a solid basis for model transformation in the metamodel approach on the one hand. On the other hand graph transformations [Roz97,EEKR99,EKMR99] provide a solid basis for model transformation. The well defined mathematical background allows clear definitions of syntax and semantics of visual languages as well as analysis techniques such as termination criteria or critical pair analysis for model transformation.

[1] Email: ehrig@cs.tu-berlin.de
[2] Email: karstene@cs.tu-berlin.de

Different model transformation approaches are published in the literature such as $ATOM^3$ [dLT04], GreAt [AKK+05], and VIATRA [VVP02] and implemented in commercial MDA tools such as ArcStyler [Arc], and XDE [Rat].

In this paper we discuss in which way the formal definition of visual languages and model transformation can be supported by typed attributed graph transformation in the sense of [HKT02,EPT04,EEPT05]. Since our main focus is on model transformation, which should be based on the abstract syntax of the source and target languages, we only consider the abstract syntax level of visual languages in this paper.

The main ideas are the following: Describing a model transformation by typed attributed graph transformation, the source and target models have to be given as typed attributed graphs. This is not a restriction, since the underlying structure of any model, especially visual models, is described best by typed attributed graphs, due to their multi-dimensional extension. Performing model transformation by typed attributed graph transformation means to take the underlying structure of a model as typed attributed graph, and to transform it according to certain transformation productions. The result is a typed attributed graph which shows the underlying structure of the target model.

A model transformation based on graph transformation can be defined by an attributed graph transformation system $GTS = (ATG, Prod)$ consisting of an attributed type graph ATG and a set of transformation productions $Prod$. The abstract syntax graphs of the source models can be specified by all (or a subset of) instance graphs over a type graph ATG_S. Correspondingly, the abstract syntax graphs of the target models are specified by all (or a subset of) instance graphs over an attributed type graph ATG_T. Both type graphs ATG_S and ATG_T have to be subgraphs of the attributed type graph ATG (see Figure 1). Starting the model transformation with instance graph AG_S typed over ATG_S, it is also typed over ATG. During the model transformation process the intermediate graphs are typed over ATG. Please note that this type graph may contain not only ATG_S and ATG_T, but also additional types, relations and attributes which are needed for the transformation process only. The result graph AG_T is automatically typed over ATG. If it is also typed over ATG_T, it fulfills one main requirement to be syntactically correct. Data types are preserved during the transformation process.

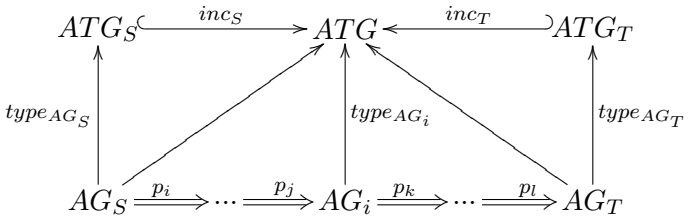


Fig. 1. Typing in the Model Transformation Process

The main ideas to define model transformations for visual languages sketched

above are defined in more detail as basic concept in section 3. This basic concept is illustrated by a small case study in section 2.

In particular, it is described which kind of correctness requirements are useful and how they can be formulated in our approach on a formal basis. In section 3 we analyse this for the basic concept discussed above. Since the basic concept has only limited expressive power we give an overview of several extensions in section 4. Expecially we discuss the use of productions with application conditions, metamodels with different kinds of constraints, generating graph grammars as well as operational semantics for the source and target languages. Moreover we analyse the consequences concerning correctness of model transformations, where according to [Tae05] correctness includes syntactical correctness, functional behavior and semantical correctness. Correctness criteria have already been surveyed by Varró and Pataricza in [VP03].

In the conclusion we summarize how far formal concepts for model transformation are supported by the theory of typed attributed graph transformation already and which problems are left for future research.

Acknowledgements

We would like to thank Gabor Karsai and Gabriele Taentzer for organizing this first international workshop on *Graph and Model Transformations* and the reviewers for several useful comments leading to an improved final version.

2 Model Transformation Case Study

In order to illustrate our basic concept to be introduced in section 3 we briefly review a model transformation from UML 2.0 statecharts to Petri nets presented in [EEdL+05,EEPT05,Gya04].

Source modelling language: Simple version of statecharts

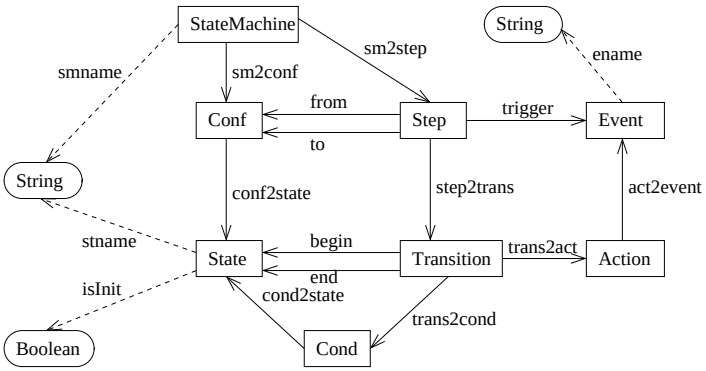


Fig. 2. Statechart Type Graph as *E-graph*

The type graph T_S of statecharts is shown in Fig. 2. In fact T_S is an *E-graph* (see [EEPT05]) where the graph nodes are represented by rectangles

and the data nodes by rounded rectangles, graph edges by solid arrows with inscription, node attribute edges by dashed arrows with inscription. In this example there are no edge attributes. The data type signature $DSIG = Sig(string) + Sig(boolean)$ is given by the signatures of strings and booleans, where only the sorts *'String'* and *'Boolean'* are shown in Fig. 2. The type graph T_S together with the final **DSIG**-algebra Z defines the attributed type graph $ATG_S = (T_S, Z)$. T_S can be considered as an extract of the metamodel of UML 2.0 statecharts. In fact, this metamodel is a proper restriction of the standard UML metamodel that explicitly introduces several notions of statecharts that are only implicitly present in the standard (such as state configurations, etc.). We consider a network of statemachines StateMachine. A single statemachine captures the behaviour of any object of a specific class by flattening the state hierarchy into *state configurations* Conf and grouping parallel transitions into Steps.

Please note, that the statechart type graph in Fig. 2 allows graphs which are non-valid statecharts. For example a Step could be connected with two StateMachines via the edge sm2step. For this reason a generating syntax grammar could be used to define precisely the source modelling language (see [EEPT05]).

Target modelling language: Petri nets

In fact, there are several variants of Petri nets in the literature. We consider place/transition nets with arc weight one and at most one token on each place, with the type graph T_T shown in Fig. 3.

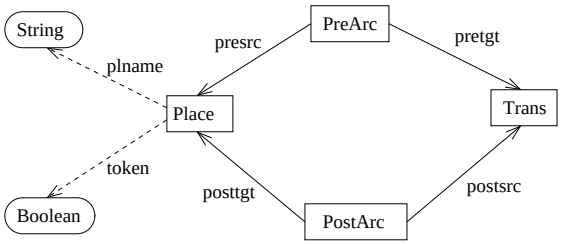


Fig. 3. Petri net type graph as *E-graph*

Reference metamodel

In order to interrelate the source and target modeling languages, we use reference metamodels [VVP02]. For instance, a reference node of type RefState (in Fig. 4) relates a source State to a target Place.

In the notation of Fig. 4 the left and right hand sides correspond to Fig. 2 and 3 respectively, where data nodes and node attributes are listed in a box below the corresponding graph node, e.g. the node attribute *'name'* and *'isInit'* of *'State'* with target *'String'* resp. *'Boolean'* in Fig. 2 is given by the attributes *'name: String'* and 'isInit: Boolean' of *'State'* in Fig. 4.

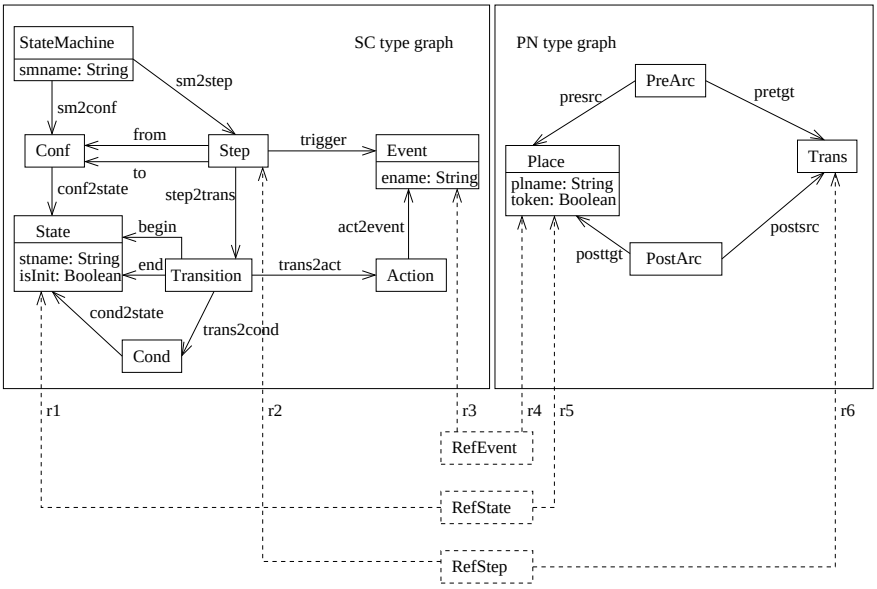


Fig. 4. Type Graph: Statecharts ⇒ Petri nets

Example Statechart: Producer-Consumer-System

As an example we will apply our model transformation to a *Producer-Consumer-System*. Fig. 5 shows the concrete syntax of the *Producer-Consumer-System* statechart in the upper part and the concrete syntax of the transformed *Producer-Consumer-System* Petri net in the lower part. The abstract syntax graph of the *Producer-Consumer-System* statechart is shown in Fig. 6 and Fig. 7 shows the abstract syntax of the transformed *Producer-Consumer-System* Petri net. Note, that Fig. 7 is typed over the Petri net type graph in Fig. 3 and Fig. 6 over the statechart type graph in Fig. 2.

Model transformation from statecharts to Petri nets

The model transformation from statecharts into Petri nets is given by 3 layers of transformation productions where Fig. 8 shows one production of each layer. We use productions in the double pushout (DPO) approach for typed attributed graphs in the sense of [EPT04,EEPT05] extended by negative application conditions (NACs) in the sense of [EEHP04]. In Fig. 8 we only show the left hand side (LHS), right hand side (RHS) and NAC (if not empty) of each rule while the interface graph is given by the intersection of LHS and RHS where common items have the same numbering. In order to get a target graph typed over the Petri net metamodel the graph of the source statecharts and the reference nodes and edges are deleted after model transformation.

The model transformation is done in the following 3 steps. In our example starting with Fig. 6 we obtain Fig. 7 as final result.

- Each statechart state is modeled with a respective place in the target Petri net model where a token in such a place denotes that the corresponding

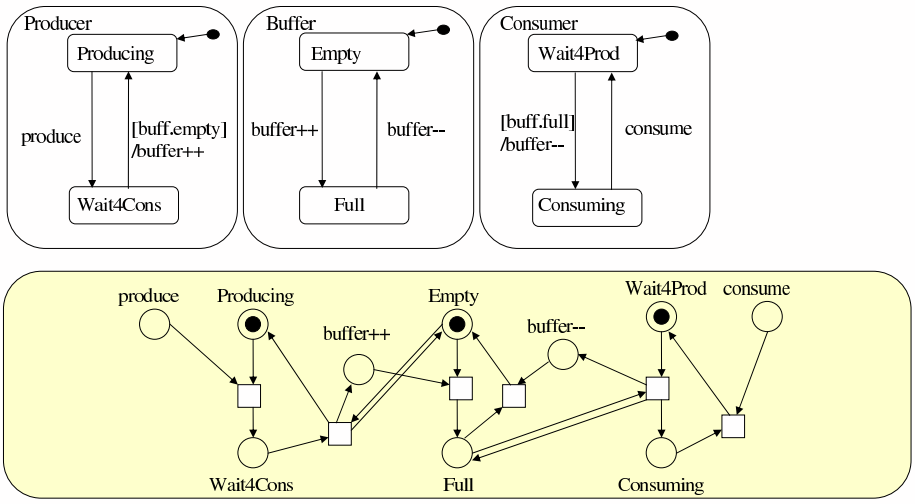


Fig. 5. Example statechart: Producer-Consumer-System Concrete Syntax Graph (Upper Part) and Concrete Syntax Graph of the Transformed Petri Net (Lower Part)

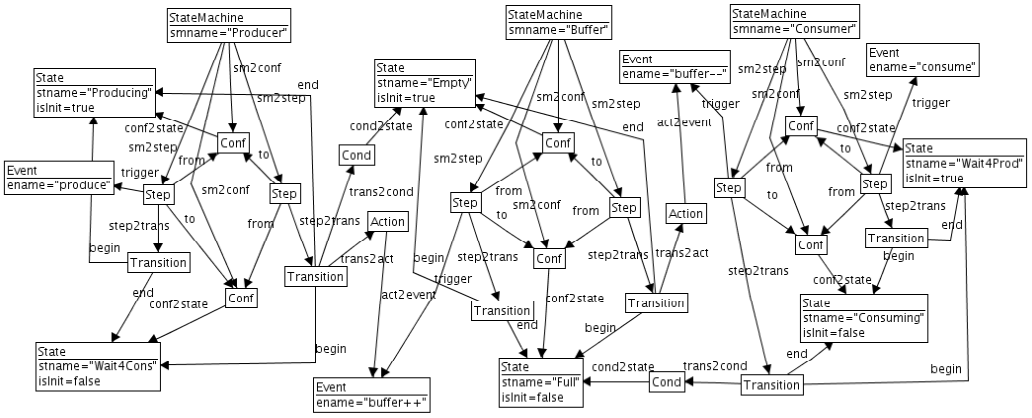


Fig. 6. Abstract Syntax Graph of Producer-Consumer-System Statechart

state is active initially (see production InitState2Place). A separate place is generated for each valid event. The negative application condition (NAC) makes sure that we can apply this rule for each state only once, because InitState2Place is applicable to a state only if the state is not yet connected to RefState by an edge with type r_1.

- Each statechart step is projected into a Petri net transition. Naturally, the Petri net should simulate how to exit and enter the corresponding states in the statechart, therefore input and output arcs of the transition should be generated accordingly (see StepFrom2PreArc). Furthermore, firing a transition should consume the token of the trigger event, and should generate tokens to (the places related to) the target (receiver) event according to the actions.

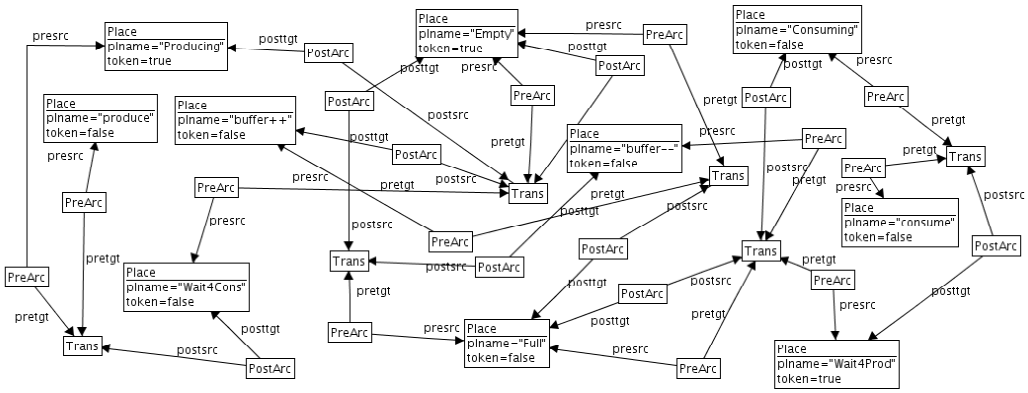


Fig. 7. Abstract Syntax Graph of the Transformed Producer-Consumer-System Petri Net

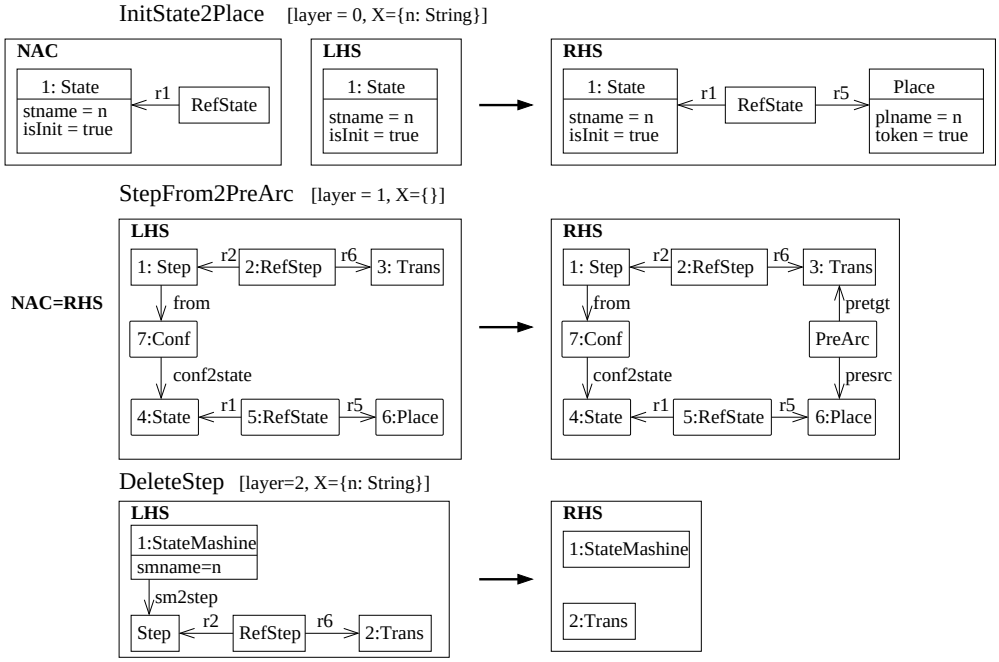


Fig. 8. Typical Transformation Productions from Layers 0, 1, 2

- Finally, we clear up the joint model by removing all model elements from the source and the reference metamodel by another set of graph transformation productions. For instance, production DeleteStep deletes a Step with a corresponding RefStep. After applying all deleting productions we obtain Fig. 7.

3 Basic Concepts of Visual Languages and Model Transformations

In this section we present our basic concept how to define visual languages and model transformations using typed attributed graph transformations. This

basic concept was sketched already in the introduction and is presented now in more detail. Especially we discuss the formal requirements for correctness of model transformation.

3.1 Visual Languages

In the basic concept studied in this section a metamodel MM consists only of an attributed type graph ATG, i.e. $MM = ATG$,
and the visual language VL is defined by all attributed graphs AG typed over ATG, written $VL = \mathbf{AGraphs_{ATG}}$. Note, that this is a very basic description of a visual language and a more elaborated one will follow in section 4.

According to [HKT02] an attributed graph $AG = (G, D)$ consists of a graph G and a data type algebra D of a given data type signature $DSIG = (S, OP)$ with the sets of sort symbols S and operation symbols OP. This allows to consider graphs with node attribution. In [EPT04] this concept is extended to allow also edge attribution, where G is not a classical graph, but an *E-graph* with edge attribute edges from graph edges to data nodes. An attributed type graph $ATG = (TG, Z)$ is an attributed graph, where TG is the graph part and Z is the final $DSIG$-algebra with $Z_s = \{s\}$ for each sort $s \in S$.

Examples of attributed type graphs $ATG = (TG, Z)$ are given in Fig. 2 - 4 in section 2, where only the graph part TG is shown. An attributed graph AG typed over ATG is given by AG together with an attributed graph morphism $type : AG \to ATG$, called type morphism. Examples of typed attributed graphs are given in Fig. 6 - 7 in section 2.

In section 5 we will consider more general cases to define visual languages corresponding to the metamodeling and the graph grammar approach respectively. In both cases, however, the definition is based on an attributed type graph ATG.

3.2 Model Transformation

The basic idea to define model transformations $MT : VL_S \to VL_T$ based on graph transformation has been discussed already in the introduction (see Fig. 1). According to 4.1 we assume to have visual languages
$VL_S = \mathbf{AGraphs_{ATG_S}}$ and $VL_T = \mathbf{AGraphs_{ATG_T}}$ based on ATG_S and ATG_T with the same data type signature $DSIG$. For model transformation we construct a new attributed type graph ATG together with data type preserving inclusions $inc_S : ATG_S \to ATG$ and $inc_T : ATG_T \to ATG$. An example for ATG, ins_S and inc_T is given in Fig. 4 in section 2, where ATG includes not only ATG_S and ATG_T, but also additional types and relations, which are needed for the model transformation. A model transformation $MT : VL_S \to VL_T$ from VL_S to VL_T based on GTS is defined by

$$MT = (VL_S, VL_T, ATG, GTS)$$

where we have in addition to VL_S, VL_T and the attributed type graph ATG an attributed graph transformation system $GTS = (ATG, Prod)$ typed over ATG with productions $Prod$ as defined in [EPT04,EEPT05]. In general $Prod$ will be given in several layers $Prod_n$ $(0 \leq n \leq k_0)$, where the productions in each layer $Prod_n$ are applied as long as possible, before going over to layer $Prod_{n+1}$ or finishing in the case $n = k_0$. This leads to a layered model transformation sequence $M_S \Rightarrow^* M_T$ via GTS where the source model M_S and the target model M_T are given by attributed graph AG_S and AG_T respectively, i.e. $M_S = AG_S$ and $M_T = AG_T$. In general the goal is to start with $AG_S \in VL_S$ and to require that the model transformation sequence $AG_S \Rightarrow^* AG_T$ via GTS leads to a unique $AG_T \in VL_T$. In this case we would have $MT(AG_S) = AG_T$ which suggests that $MT : VL_S \to VL_T$ has functional behavior. In general, however, we cannot be sure to achieve this goal unless we make sure that suitable correctness conditions are satisfied, which will be discussed below.

An example for a model transformation MT is given in section 2, where the type graph inclusions are shown in Fig. 4 and typical examples of productions $Prod_0$, $Prod_1$ and $Prod_2$ in Fig. 8. Note that most of these productions have negative application conditions, which are discussed in section 4.1 as an extension of the basic concepts.

3.3 Correctness Requirements

In the introduction we have pointed out already that in general we have the following three kinds of correctness requirements:

Syntactical Correctness, Functional Behavior and Semantical Correctness.

3.3.1 Syntactical Correctness in the Basic Concept

In our basic concept studied in this section syntactical correctness of the model transformation $MT : VL_S \to VL_T$ based on GTS means that for each $AG_S \in VL_S$ there is a model transformation sequence $AG_S \Rightarrow^* AG_T$ via GTS with $AG_T \in VL_T$, i.e. AG_T is typed over ATG_T.

Up to now, however, there are no general results or techniques to assure syntactical correctness, unless we have already $ATG_T = ATG$ or we enforce $AG_T \in VL_T$ by taking the restriction of the resulting graph to ATG_T. This, however, may not be appropriate from the semantical point of view. Otherwise a necessary condition for syntactical correctness in the case of $ATG_T \not\subseteq ATG$ is that for each label in $ATG \setminus ATG_T$ there is a deleting production, provided that all labels in ATG are used in the generating productions. This condition is satisfied for the case study in section 2. If it is possible to show syntactical correctness of MT seperately for each layer – with distinguished intermediate type graphs – this would certainly imply syntactical correctness of MT by composition.

3.3.2 Functional Behavior

Functional behavior of the model transformation $MT : VL_S \to VL_T$ based on GTS means that we have local confluence and termination of GTS which implies that MT defines a function from VL_S to VL_T. Local confluence can be shown by the *Local Church-Rosser Theorem* for parallel independent direct transformations. Critical pair analysis allows to analyse all parallel dependencies and we obtain local confluence if we have strict confluence of all critical pairs. This result is called *Local Confluence Theorem*, also known as *Critical Pair Lemma*, and has been shown for typed attributed graph transformation in [EPT04]. However, up to now this result is only known for productions without negative application conditions (NACs). Hence presently this result is not applicable to our case study, where most of the productions have NACs.

As mentioned in section 2 a generating syntax grammar has been provided in [EEPT05] which allows to conclude that the model transformation in our case study is locally confluent, provided that the model transformation source language is restricted to statecharts generated by the syntax grammar.

Termination critera for layered graph transformations have been shown in [EEdL+05,EEPT05] (see also [KHE03]) for layered labeled and typed attributed graph transformation systems and verified for our case study already in these papers. Altogether our case study – restricted to the generating syntax grammar for statecharts as given in [EEPT05] – has functional behavior.

3.3.3 Semantical Correctness in Basic Case

In the basic case we have no explicit behavior for the source and target languages VL_S and VL_T. This means that there is no explicit way to show that the model transformation MT is behavior preserving and hence semantically correct. In our case study we also have no explicit behavior for models of the source and target language. Hence we are only able to check semantical compatibility of the given state chart and the transformed Petri net in Fig. 5 on an intuitive basis, or to extend the source and/or the target language by operational semantics (see section 4.4).

If only the target language has a formal semantics we can define a formal semantics for the source language via the model transformation. The case that both languages have an operational semantics is studied in section 4.4.

4 Overview of Extended Formal Concepts

In this section we discuss several formal extensions of the basic concepts for visual languages and model transformations. We present each extension separately and discuss specific consequences concerning correctness of model transformations. For most of the application domains of model transformations it makes sense to consider several of these extensions simultaneously.

4.1 Model Transformation based on Graph Transformation Systems with Application Conditions

In section 3 we have assumed for the basic concept of model transformations $MT : VL_S \to VL_T$ that MT is defined by a typed attributed graph transformation system GTS using productions without application conditions, because main parts of the theory in [EPT04,EEPT05] are only avaliable for this basic case. In our case study in section 2 , however, we have already used and explained negative application conditions (NACs). In fact, NACs are the most common kind of application conditions, but not the most general ones. In [EEHP04,EEPT05] we have defined more general kinds of positive and negative application conditions. One of the main result shows that for each graph constraint *constr* there is an equivalent application condition $appl(constr)$. This means that for each direct transformation $G \Rightarrow H$ *via* (p, m), where the match m satisfies the application condition $appl(constr)$ the resulting graph H satisfies the graph constraint *constr*.

This extension has no direct consequences for syntactical correctness in the basic concept, but it is important when we consider visual languages VL_S and VL_T defined by type graphs and constraints (see 4.2). Concerning functional behavior this extension supports to show termination, because the termination criteria in [EEdL+05,EEPT05] assume for the nondeletion layers that we have productions with NACs. Concerning local confluence, however, this extension causes problems, because the local confluence theorem in [EEHP04,EEPT05] has to be extended to productions with application conditions. Even for NACs this seems to be a nontrivial task. The consequences of this extension for semantical correctness have to be discussed in connection with extension 4.4.

4.2 Visual Languages based on Metamodels with Constraints

In the basic concept we have assumed that the visual languages VL_S and VL_T of the model transformation $MT : VL_S \to VL_T$ are completely defined by attributed type graphs ATG_S and ATG_T respectively. In the metamodel approach for visual languages, however, the metamodels are defined by class diagrams CD_S and CD_T respectively. These metamodels correspond roughly to our type graphs ATG_S and ATG_T.

The models defining VL_S and VL_T in the metamodel approach, however, are defined not only by the class diagrams, but they are restricted by suitable OCL constraints. This motivates to extend our concept of metamodels for visual languages VL also by constraints, i.e. $MM(ATG, Constr)$, where $Constr$ is a suitable set of constraints. In the theory of typed attributed graph transformations there exists already the concept of graph constraints (see [HW95,EEHP04,EEPT05]) which especially allow to express multiplicity constraints in class diagrams (see [RT05]). An atomic (positive) constraint $PC(a)$ consists of a morphism $a : P \to C$, and a general graph constraint is a Boolean formula over atomic graph constraints. A graph G satisfies $PC(a)$

if for every injective graph morphism $p : P \to G$ there is an injective graph morphism $q : C \to G$ s.t. $q \circ a = p$. In addition to graph constraints we can consider data type constraints given by equations or first order formulas over the data type signature $DSIG$ of ATG_S and ATG_T. An attributed graph $AG = (G, D)$ satisfies a data type constraint *data-constr*, if the $DSIG$-algebra D satisfies *data-constr* (see [EM85] for the case of equations). On the one hand data type constraints can be used as global invariants for all attributed graphs AG in the language and on the other hand as application conditions for rules. In this case the rule can only be applied to an attributed graph AG, if AG satisfies the data type constraint of the rule. In such a data type constraint we could require that certain attributes of AG satisfy specific conditions.

Although some kinds of OCL constraints are expressible by graph and data type constraints already, there is no straight forward way to express general OCL constraints using these constraints only. For this reason we propose to define in addition to graph and data type constraints in analogy to OCL constraints of UML 2.0 a new kind of constraints for typed attributed graphs, called *graph-OCL constraints*. In [EE05] we present first ideas how to define syntax and satisfaction for such constraints. Using these ideas a metamodel $MM = (ATG, Constr)$ may include three kinds of constraints $Constr = (Constr_1, Constr_2, Constr_3)$, where $Constr_1$, $Constr_2$, and $Constr_3$ are sets of graph, data type, and graph-OCL constraints respectively. Presently it is open which kind of constraints to use for specific visual languages.

In the extended version a visual language VL with metamodel $MM = (ATG, Constr)$ consists of all attributed graphs typed over ATG, which satisfy all the constraints in $Constr$. This extension has important consequences for the syntactical correctness of a model transformation $MT : VL_S \to VL_T$, because we have to require for each $AG_S \in VL_S$ not only the existence of a model transformation $AG_S \Rightarrow^* AG_T$ via GTS where AG_T is typed over ATG_T, but also that AG_T satisfies the constraints $Constr_T$ of the metamodel $MM_T(ATG_T, Constr_T)$ for the target language VL_T. Since $AG_S \in VL_S$ satisfies the constraints $Constr_S$ of the source language we have to show for syntactical correctness of MT that the corresponding transformation sequences $AG_S \Rightarrow^* AG_T$ are compatible with $Constr_S$ and $Constr_T$. For graph constraints there are already some techniques to assure satisfaction by transforming graph constraints into corresponding application conditions (see [HW95,EEHP04,EEPT05]). It is open how to use this in the context of model transformations. For data type constraints as global invariants there is no problem in the basic case, where the inclusions $inc_S : ATG_S \to ATG$ and $inc_T : ATG_T \to ATG$ are data type preserving. For rule constraints and for global constraints in the case of data type sensitive extensions (see 4.5.3) well-known techniques for algebraic specifications in [EM85] can support the verification process. How to handle graph-OCL constraints in this context, however, is completely open and is certainly an interesting research topic in connection with graph-OCL constraints. For functional behavior and seman-

tical correctness this extension has no direct consequences as long as the graph transformation system of the model transformation is not changed.

4.3 Visual Languages based on Generating Graph Grammars

Another alternative to define visual languages are graph grammar approaches like DiaGen [MV95] or GenGED [Bar98], where the visual language VL is given by all graphs which can be generated by the corresponding graph grammar. As extension of the basic case in section 3.1 we consider now the case that a visual language VL is generated by a typed attributed graph grammar $GG = (ATG, Prod, AG_0)$, where $Prod$ is a set of productions and AG_0 an attributed start graph, both typed over ATG (see [HKT02,EPT04,EEPT05]). This means that the visual language VL is given by

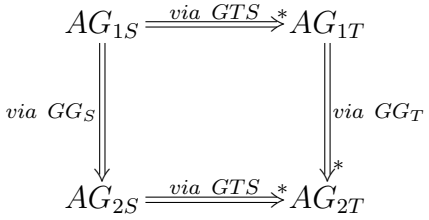
$$VL = \{AG | AG_0 \Rightarrow^* AG \text{ via } Prod\} \subseteq \mathbf{AGraphs_{ATG}}$$

For the model transformation $MT : VL_S \to VL_T$ considered in section 3 we assume now the extension that VL_S and VL_T are generated by typed attributed graph grammars $GG_S = (ATG_S, Prod_S, AG_{0S})$ and $GG_T = (ATG_T, Prod_T, AG_{0T})$ respectively.

This extension of the basic concepts has the following consequences for syntactical correctness of $MT : VL_S \to VL_T$ with $MT = (VL_S, VL_T, ATG, GTS)$ based on $GTS = (ATG, Prod)$:

First of all we start the model transformation not with any graph $AG_S \in VL_S$, but with the start graph AG_{0S}. To solve the *Initialization Problem* means to construct for $AG_{0S} \in VL_S$ a transformation sequence $AG_{0S} \Rightarrow^* AG_T$ via GTS such that there is also a generating transformation sequence $AG_{0T} \Rightarrow^* AG_T$ via GG_T which implies $AG_T \in VL_T$. For every other $AG_S \in VL_S$ we have by definition of VL_S a generating transformation sequence $AG_{0S} \Rightarrow^* AG_S$ via GG_S. In order to show syntactical correctness also for AG_S we propose to solve the following *Mixed Confluence Problem*:

Given $AG_{1S} \Rightarrow^* AG_{1T}$ via GTS with $AG_{1S} \in VL_S$ and $AG_{1T} \in VL_T$ and a direct transformation $AG_{1S} \Rightarrow AG_{2S}$ via GG_S we have to construct transformation sequences $AG_{2S} \Rightarrow^* AG_{2T}$ via GTS and $AG_{1T} \Rightarrow^* AG_{2T}$ via GG_T leading to the following *Mixed Confluence Diagram*:



Let us recall that the model transformation $MT : VL_S \to VL_T$ based on GTS is syntactically correct, if for each $AG_S \in VL_S$ there is a transformation sequence $AG_S \Rightarrow^* AG_T$ via GTS with $AG_T \in VL_T$. This leads to the

following result.

Fact 4.1 (Syntactical Correctness of Model Transformation) *Given visual languages VL_S and VL_T generated by graph grammars GG_S and GG_T respectively then a model transformation $MT : VL_S \to VL_T$ based on GTS is syntactically correct, if the* Initialization *and the* Mixed Confluence Problem *can be solved.*

Proof. Given $AG_S \in VL_S$ we have by definition of VL_S a generating transformation sequence $AG_{0S} \Rightarrow^* AG_S$ via GG_S. Solving the *Initialization Problem* leads to a transformation sequence $AG_{0T} \Rightarrow^* AG'_T$ via GG_T with $AG'_T \in VL_T$. Since also the *Mixed Confluence Problem* can be solved: We have for each direct transformation $AG_{1S} \Rightarrow AG_{2S}$ via GG_S of the transformation sequence $AG_{0S} \Rightarrow^* AG_S$ via GG_S a single *Mixed Confluence Diagram* leading by composition to the following composed *Mixed Confluence Diagram*:

$$\begin{array}{ccc} AG_{0S} & \overset{via\ GTS\ *}{\Longrightarrow} & AG'_T \\ {\scriptstyle via\ GG_S}\ \Big\Downarrow{\scriptstyle *} & & \Big\Downarrow{\scriptstyle *}\ {\scriptstyle via\ GG_T} \\ AG_S & \overset{via\ GTS\ *}{\Longrightarrow} & AG_T \end{array}$$

Finally $AG'_T \in VL_T$ and $AG'_T \Rightarrow^* AG_T$ via GG_T implies $AG_T \in VL_T$ s.t. the bottom sequence $AG_S \Rightarrow^* AG_T$ via GTS implies syntactical correctness.□

Of course, it remains to find suitable techniques to solve the *Initialization* and the *Mixed Confluence Problem*. For the second problem it may be possible to use the techniques of critical pair analysis to show local confluence for typed attributed graph transformation systems (see [HKT02,EPT04,EEPT05]).

4.4 Visual Languages with Operational Semantics

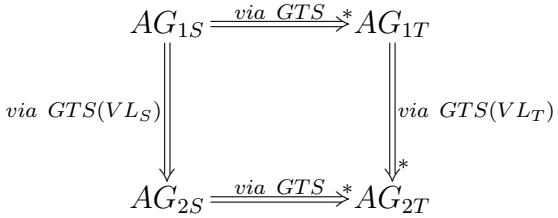
Up to now we have only considered visual languages from the syntactical point of view. Now we extend the basic concept of visual languages VL taking into account an operational semantics for VL. According to [Tae05] this means that we assume to have a simulation specification. In our context – where VL is typed over ATG – it makes sense to define the simulation specification by a typed attributed graph transformation system $GTS(VL) = (ATG, Prod_{Sim})$ with same ATG, where $Prod_{Sim}$ are the productions for the simulation of VL. The operational semantics of VL in this case is given by

$$Sim(VL) = \{AG \Rightarrow^* AG' | AG, AG' \in VL \ \&\ AG \Rightarrow^* AG' \ via\ GTS(VL)\}$$

where each $AG \Rightarrow^* AG'$ in $Sim(VL)$ can be considered as a simulation scenario of VL (see [Tae05]). For the model transformation $MT : VL_S \to VL_T$ considered in section 4 we assume now the extension that we have VL_S and VL_T with operational semantics based on typed attributed graph transformation systems $GTS(VL_S) = (ATG_S, Prod_S)$ and $GTS(VL_T) =$

$(ATG_T, Prod_T)$ respectively.

This extension of the basic concept has of course consequences for the semantical correctness of $MT : VL_S \to VL_T$. More precisely this extension allows first of all to define semantical correctness on a formal basis in the following way: The model transformation $MT : VL_S \to VL_T$ based on GTS with operational semantics of VL_S and VL_T based on $GTS(VL_S)$ and $GTS(VL_T)$ respectively is called *semantically correct*, if for each transformation sequence $AG_{1S} \Rightarrow^* AG_{1T}$ via GTS with $AG_{1S} \in VL_S$ and $AG_{1T} \in VL_T$ and each simulation step $AG_{1S} \Rightarrow AG_{2S}$ via $GTS(VL_S)$ there is a transformation sequence $AG_{2S} \Rightarrow^* AG_{2T}$ via GTS and a simulation sequence $AG_{1T} \Rightarrow^* AG_{2T}$ via $GTS(VL_T)$ leading to the following *Mixed Confluence Diagram*:



It is interesting to note that the *Mixed Confluence Diagram* for semantical correctness above is formally very similar to the *Mixed Confluence Diagram* for syntactical correctness in section 4.3 . In fact, only the graph grammars GG_S and GG_T are replaced by the graph transformation system $GTS(VL_S)$ and $GTS(VL_T)$ respectively. This means that formal techniques to be developed which are suitable to solve the *Mixed Confluence Problem* can be applied to show syntactical as well as semantical correctness.

Finally let us note that it may be suitable for some kind of application to relax the condition for semantical correctness. It may be the case that not each single simulation step $AG_{1S} \Rightarrow AG_{2S}$ via $GTS(VL_S)$ is semantically meaningful, but only suitable simulation sequences $AG_{1S} \Rightarrow^* AG_{2S}$ via $GTS(VL_S)$. In this case we would have to replace the single step in the *Mixed Confluence Diagram* by those sequences $AG_{1S} \Rightarrow^* AG_{2S}$ via $GTS(VL_S)$ which are semantically meaningful.

4.5 Other Formal Extensions

In addition to the extension of the basic concept discussed above let us briefly mention some other formal extensions. In these cases, however, it will be even more difficult to analyse the consequences for correctness of corresponding model transformations.

4.5.1 Attributed Type Graphs with Inheritance

The attributed type graphs ATG_S, ATG_T and ATG used in the basic concept are replaced by attributed type graphs with inheritance as introduced in [BEE+05]. This is motivated by the concept of class inheritance in the object-oriented paradigm in general and in particular by the concept of inheritance

in the UML metamodel [UML] and allows much more efficient representation of typed attributed graph transformation systems and grammars and hence of syntax and semantics of visual languages and model transformations.

4.5.2 View-based Approach for Visual Languages

In the basic concept of section 3.1 we have assumed that the metamodel of a visual language VL consists only of one attributed type graph ATG. Similar to the case of UML we consider now different views of VL which are defined by suitable restrictions of the metamodel. More precisely we discuss now a view-based approach for a visual language VL, where VL is represented by different views $V_1, ..., V_n$, and each view V_i is represented by an attributed type graph $ATG_i \subseteq ATG \quad (i = 1, .., n)$. If VL_S and VL_T are both replaced by different views it would make sense to represent also a model transformation $MT : VL_S \to VL_T$ by different views, i.e. model transformations $MT_i : VL_{Si} \to VL_{Ti}$. In this case well-known problems concerning consistency of views for visual languages would imply also consistency problems for the views MT_i of the model transformation MT.

4.5.3 Data Type Sensitive Type Graph Extension

Instead of data type preserving type graph inclusions $inc_S : AGT_S \to AGT$ and $inc_T : AGT_T \to AGT$ considered in the basic concept, we allow data type sensitive type graph inclusions. This means that we also allow different data type signatures $DSIG_S$, $DSIG$, and $DSIG_T$ with inclusions $DSIG_S \subseteq DSIG$ and $DSIG_T \subseteq DSIG$. From the practical point of view this seems reasonable, because in general the data types of different visual languages VL_S and VL_T will also be different. From the theoretical point of view, however, this extension will cause some problems because the current theory of typed attributed graph transformation systems in [EPT04,EEPT05] is essentially based on a class $\mathcal{M}$ of attributed graph morphisms which are injective and data type preserving.

5 Conclusion

The concept of typed attributed graph transformation is a powerful technique to model not only visual languages, but also to define model transformations between visual languages. This includes not only the formal definition of these concepts, but also correctness of model transformations which is certainly an important research topic. Concerning correctness we distinguish between syntactical correctness, functional behavior and semantical correctness.

According to the basic concept of visual languages and model transformations presented in this paper visual languages are defined by metamodels consisting of attributed type graphs only and model transformations by typed attributed graph transformation systems. In a case study (see section 2) we show how to define such a model transformation from a simple version of state

charts to Petri nets.

The theory of typed attributed graph transformation known up to now (see [HKT02,EPT04,EEPT05]) supports to show functional behavior for model transformations according to the basic concept discussed above. In fact, the local confluence theorem based on critical pairs and the termination criteria in [EEdL+05] allow to verify confluence and termination of the corresponding typed attributed graph transformation system.

The basic concept, however, allows only to model simple cases of visual languages and model transformation. For this reason we have discussed several extensions of the basic concept including application conditions, constraints, generating graph grammars and operational semantics. Especially we propose different kinds of constraints including not only graph and data type constraints, which have been studied in the literature [HW95,EEHP04,EEPT05,EM85], but also in analogy to OCL constraints of UML a new kind of constraints for typed attributed graphs, called graph-OCL constraints (see [EE05]). We claim that an approach of visual languages and model transformations based on attributed type graphs and all these constraints has the potential to become at least as powerful as most of the metamodeling approaches known in the literature [dLT04,AKK+05,VVP02].

In this paper we give an overview in which way the different extensions of the basic concept influence the correctness criteria for model transformations. The current theory of typed attributed graph transformations is not yet ready to support the verification of most of these correctness criteria in the extended concepts. But at least it is a very good basis and provides clear indications for future research concerning these issues.

References

[AKK+05] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, and Vizhanyo A. The design of a simple language for graph transformations, journal in software and system modeling. *Journal on Software and System Modeling*, 2005. in review.

[Arc] ArcStyler. http://www.arcstyler.com.

[Bar98] R. Bardohl. GenGEd - A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars. In *Proc. IEEE Symposium on Visual Languages (VL'98)*, pages 48–55, 1998.

[BEE+05] R. Bardohl, H. Ehrig, K. Ehrig, J. de Lara, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. In *Technical Report 2005/3*. Technical University Berlin, 2005. Short version to appear in Proceedings VLHCC'05.

[dLT04] J. de Lara and G. Taentzer. Automated model transformation and its validation with atom^3 and agg. *In DIAGRAMS'2004 (Cambridge, UK).*

Lecture Notes in Artificial Intelligence 2980, Springer, 2, 2004. pp.: 182–198.

[EE05] H. Ehrig and K. Ehrig. Towards Graph-OCL-Constraints. Technical report, Technical Report, TU Berlin, to appear, 2005.

[EEdL+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 214–228. Springer Verlag, 2005.

[EEHP04] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and Application Conditions: From Graphs to High-Level Structures. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proceedings of ICGT 2004*, volume 3256 of *LNCS*, pages 287–303. Springer, 2004.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol 2: Applications, Languages and Tools.* World Scientific, 1999.

[EEPT05] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs in TCS, Springer to appear, 2005.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol 3: Concurrency, Parallelism and Distribution.* World Scientific, 1999.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* EATCS Monographs in TCS. Springer, 1985.

[EPT04] H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proceedings of ICGT 2004*, volume 3256 of *LNCS*, pages 161–177. Springer, 2004.

[Gya04] Szilvia Gyapay. Model transformation from general resource models to petri nets using graph transformation. In *Technical Report 2004/19.* Technical University Berlin, 2004.

[HKT02] R. Heckel, J. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of ICGT 2002*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.

[HW95] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Rewriting - a Constructive Approach. In *Proceedings of SEGRAGRA 1995*, volume 2 of *ENTCS*, 1995.

[KHE03] J. M. Küster, R. Heckel, and G. Engels. Defining and Validating Transformations of UML Models. In J. Hosking and P. Cox, editors, *IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003) - Auckland, October 28 - October 31 2003, Auckland, New Zealand, Proceedings*, pages 145–152. IEEE Computer Society, 2003.

[MV95] M. Minas and G. Viehstaedt. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *Proc. IEEE Symp. on Visual Languages*, pages 203–210, Darmstadt, Germany, September, 5-9 1995.

[OMGa] Object Management Group – OMG. Meta-Object Facility (MOF). http://www.omg.org/mof.

[OMGb] Object Management Group – OMG. Model Driven Architecture (MDA). http://www.omg.org/mda.

[Rat] IBM Rational. XDE Developer. http://www.rational.com/products/xde.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol 1: Foundations*. World Scientific, 1997.

[RT05] A. Rensink and G. Taentzer. Ensuring structural constraints in graph-based models with type in heritance. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.

[Tae05] G. Taentzer. Graph Transformation in Software Engineering, Internal Report TU-Berlin, 2005.

[UML] UML. Unified Modeling Language – Version 2.0. http://www.uml.org.

[VP03] D. Varró and A. Pataricza. Automated formal verification of model transformations. In Jan Jürjens, Bernhard Rumpe, Robert France, and Eduardo B. Fernandez, editors, *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, pages 63–78. Technische Universität München, September 2003.

[VVP02] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):pp. 205–227, 2002.